Case study

Inventory Management System for B2B SaaS

Submitted by: Asita Sonal
Date: 04-Dec-2025

PART 1: CODE REVIEW & DEBUGGING

Original Code:

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.json

    # Create new product
    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=data['price'],
        warehouse_id=data['warehouse_id']
    )

    db.session.add(product)
    db.session.commit()

    # Update inventory count
    inventory = Inventory(
        product_id=product.id,
        warehouse_id=data['warehouse_id'],
        quantity=data['initial_quantity']
    )

    db.session.add(inventory)
    db.session.commit()

    return {"message": "Product created", "product_id": product.id}
```

Case study

A. Issues Identified

Issue	Description
Missing Input Validation	• No validation for required fields or data types • No SKU uniqueness check
Poor Error Handling	• No try-catch blocks for database operations • No handling for missing/invalid data
Transaction Management Issues	• Two separate commits create risk of partial data • No rollback mechanism if second operation fails
Missing Authentication/Authorization	• No verification that user has permission to create products • No company_id linking products to companies
SQL Injection Vulnerability	• Direct use of user input without sanitization (ORM reduces but does not eliminate risk)
No Duplicate SKU Prevention	• SKU must be unique but no constraint or check before insert
Missing Response Codes	• Always returns success even on failure • No proper HTTP codes like **201**, **400**, **500**
Business Logic Gaps	• Creates inventory without checking warehouse–company mapping • No product category/type for threshold calculations

Missing Input Validation

a. No validation for required fields or data types
b. No SKU uniqueness check

2. **Poor Error Handling**
   a. No try-catch blocks for database operations
   b. No handling for missing/invalid data
3. **Transaction Management Issues**
   a. Two separate commits create risk of partial data
   b. No rollback mechanism if second operation fails
4. **Missing Authentication/Authorization**
   a. No verification that user has permission to create products
   b. No company_id linking products to specific companies
5. **SQL Injection Vulnerability**
   a. Direct use of user input without sanitization (though ORM helps)
6. **No Duplicate SKU Prevention**
   a. SKUs must be unique but no constraint check before insert
7. **Missing Response Codes**
   a. Always returns success, even on failure
   b. No proper HTTP status codes (201, 400, 500)
8. **Business Logic Gap**
   a. Creates inventory but doesn't validate warehouse belongs to company
   b. No product category/type for threshold calculations

Case study

B. Impact Analysis

Issue	Impact	Example	Business Impact
1. Missing Input Validation	Server crashes with KeyError if required fields are missing	{ "name": "Widget" } → SKU missing → API returns 500	Poor user experience, difficult debugging
2. Poor Error Handling	Unhandled exceptions crash endpoint, leaking stack traces	Database connection fails → whole request fails	System downtime, exposure of internal errors
3. Transaction Management Issues	Product saved but inventory creation fails → inconsistent data	DB crash between commits → orphaned product	Incorrect inventory records, broken tracking
4. Missing Authorization	Unauthorized users can create products in any warehouse	User from Company A creates product in Company B's warehouse	Security breach, cross-tenant data corruption
5. SQL Injection (Low Risk)	Direct field use still risky if ORM is bypassed	Malicious script in product name	Potential data breach and system compromise
6. No Duplicate SKU Prevention	Duplicate SKU causes DB constraint failure or silent errors	Two users submit same SKU simultaneously	Confusion, blocked product creation flows
7. Missing Response Codes	API always returns 200 even when failing	Validation fails → still returns success	Frontend displays incorrect state, user confusion
8. Business Logic Gap	Product can be linked to wrong warehouse/company	Product from Company A assigned to B's warehouse	Multi-tenant data leakage, compliance issues

Corrected Code:

```
from flask import request, jsonify

from sqlalchemy.exc import IntegrityError, SQLAlchemyError

from decimal import Decimal, InvalidOperation

@app.route('/api/products', methods=['POST'])

def create_product():

"""

Create a new product with initial inventory.
```

Case study

Required fields: name, sku, price, warehouse_id, initial_quantity

Optional fields: description, category, low_stock_threshold

"""

try:

data = request.json

Validate required fields

required_fields = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']

missing_fields = [field for field in required_fields if field not in data]

if missing_fields:

return jsonify({

"error": "Missing required fields",

"missing": missing_fields

}), 400

Validate data types and ranges

try:

price = Decimal(str(data['price']))

if price < 0:

return jsonify({"error": "Price must be non-negative"}), 400

except (InvalidOperation, ValueError):

return jsonify({"error": "Invalid price format"}), 400

initial_quantity = data['initial_quantity']

if not isinstance(initial_quantity, int) or initial_quantity < 0:

return jsonify({"error": "Initial quantity must be non-negative integer"}), 400

Validate SKU format (alphanumeric and hyphens only)

sku = data['sku'].strip()

if not sku or len(sku) > 50:

Case study

```
return jsonify({"error": "SKU must be 1-50 characters"}), 400

# Check if SKU already exists

existing_product = Product.query.filter_by(sku=sku).first()

if existing_product:

return jsonify({

"error": "SKU already exists",

"existing_product_id": existing_product.id

}), 409

# Verify warehouse exists and get company_id

warehouse = Warehouse.query.get(data['warehouse_id'])

if not warehouse:

return jsonify({"error": "Warehouse not found"}), 404

# TODO: Add authentication check here

# Verify current user has permission for this warehouse's company

# if not has_permission(current_user, warehouse.company_id):

# return jsonify({"error": "Unauthorized"}), 403

# Begin transaction - both product and inventory must succeed

try:

# Create new product

product = Product(

name=data['name'].strip(),

sku=sku,

price=price,

company_id=warehouse.company_id, # Link to company

category=data.get('category', 'general'), # For threshold logic

low_stock_threshold=data.get('low_stock_threshold', 10),
```

Case study

```
description=data.get('description', '')

)

db.session.add(product)

db.session.flush() # Get product.id without committing

# Create initial inventory record

inventory = Inventory(

product_id=product.id,

warehouse_id=data['warehouse_id'],

quantity=initial_quantity,

last_updated=db.func.now()

)

db.session.add(inventory)

# Log inventory change for audit trail

inventory_log = InventoryHistory(

product_id=product.id,

warehouse_id=data['warehouse_id'],

quantity_change=initial_quantity,

new_quantity=initial_quantity,

change_type='initial_stock',

changed_by=None, # TODO: Add current_user.id

notes='Initial product creation'

)

db.session.add(inventory_log)

# Commit all changes atomically

db.session.commit()

return jsonify({
```

Case study

"message": "Product created successfully",

"product_id": product.id,

"sku": product.sku,

"warehouse_id": data['warehouse_id'],

"initial_quantity": initial_quantity

}), 201

except IntegrityError as e:

db.session.rollback()

Handle database constraint violations

return jsonify({

"error": "Database constraint violation",

"details": str(e.orig)

}), 409

except SQLAlchemyError as e:

db.session.rollback()

Log error for debugging

app.logger.error(f"Database error creating product: {str(e)}")

return jsonify({

"error": "Database error occurred",

"message": "Please try again later"

}), 500

except Exception as e:

Catch any unexpected errors

app.logger.error(f"Unexpected error in create_product: {str(e)}")

return jsonify({

"error": "An unexpected error occurred",

"message": "Please contact support if this persists"

}), 500

Key Improvements:

- Comprehensive input validation with clear error messages
- Proper HTTP status codes (201, 400, 404, 409, 500)
- Single transaction with rollback capability
- SKU uniqueness check before insert
- Warehouse existence validation
- Company-product linking for multi-tenancy
- Audit trail with InventoryHistory
- Decimal handling for precise price calculations
- Optional fields with sensible defaults
- Error logging for debugging

PART 2: DATABASE DESIGN

A. Database Schema

-- Companies table (multi-tenant architecture)

CREATE TABLE companies (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

is_active BOOLEAN DEFAULT TRUE,

subscription_tier VARCHAR(50) DEFAULT 'basic'

);

CREATE INDEX idx_companies_is_active ON companies(is_active);

-- Warehouses table

CREATE TABLE warehouses (

id SERIAL PRIMARY KEY,

Case study

```
company_id INTEGER NOT NULL REFERENCES companies(id) ON DELETE CASCADE,
name VARCHAR(255) NOT NULL,
address TEXT,
city VARCHAR(100),
state VARCHAR(100),
country VARCHAR(100),
postal_code VARCHAR(20),
is_active BOOLEAN DEFAULT TRUE,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
CONSTRAINT unique_warehouse_name_per_company UNIQUE(company_id, name)
);
CREATE INDEX idx_warehouses_company_id ON warehouses(company_id);
CREATE INDEX idx_warehouses_is_active ON warehouses(is_active);
-- Suppliers table
CREATE TABLE suppliers (
id SERIAL PRIMARY KEY,
company_id INTEGER NOT NULL REFERENCES companies(id) ON DELETE CASCADE,
name VARCHAR(255) NOT NULL,
contact_name VARCHAR(255),
contact_email VARCHAR(255),
contact_phone VARCHAR(50),
address TEXT,
payment_terms VARCHAR(100),
lead_time_days INTEGER DEFAULT 7,
is_active BOOLEAN DEFAULT TRUE,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

Case study

```
CONSTRAINT unique_supplier_name_per_company UNIQUE(company_id, name)

);

CREATE INDEX idx_suppliers_company_id ON suppliers(company_id);

CREATE INDEX idx_suppliers_is_active ON suppliers(is_active);

-- Products table

CREATE TABLE products (

id SERIAL PRIMARY KEY,

company_id INTEGER NOT NULL REFERENCES companies(id) ON DELETE CASCADE,

sku VARCHAR(50) NOT NULL UNIQUE,

name VARCHAR(255) NOT NULL,

description TEXT,

category VARCHAR(100) DEFAULT 'general',

price DECIMAL(12, 2) NOT NULL CHECK (price >= 0),

cost DECIMAL(12, 2) CHECK (cost >= 0),

low_stock_threshold INTEGER DEFAULT 10 CHECK (low_stock_threshold >= 0),

is_bundle BOOLEAN DEFAULT FALSE,

is_active BOOLEAN DEFAULT TRUE,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);

CREATE INDEX idx_products_company_id ON products(company_id);

CREATE INDEX idx_products_sku ON products(sku);

CREATE INDEX idx_products_category ON products(category);

CREATE INDEX idx_products_is_active ON products(is_active);

-- Product-Supplier relationship (many-to-many)

CREATE TABLE product_suppliers (
```

Case study

```
id SERIAL PRIMARY KEY,
product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
supplier_id INTEGER NOT NULL REFERENCES suppliers(id) ON DELETE CASCADE,
supplier_sku VARCHAR(100),
cost DECIMAL(12, 2),
lead_time_days INTEGER DEFAULT 7,
minimum_order_quantity INTEGER DEFAULT 1,
is_preferred BOOLEAN DEFAULT FALSE,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
CONSTRAINT unique_product_supplier UNIQUE(product_id, supplier_id)
);
CREATE INDEX idx_product_suppliers_product_id ON product_suppliers(product_id);
CREATE INDEX idx_product_suppliers_supplier_id ON product_suppliers(supplier_id);
-- Inventory table (tracks current stock levels)
CREATE TABLE inventory (
id SERIAL PRIMARY KEY,
product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
warehouse_id INTEGER NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
quantity INTEGER NOT NULL DEFAULT 0 CHECK (quantity >= 0),
reserved_quantity INTEGER DEFAULT 0 CHECK (reserved_quantity >= 0),
last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
last_counted_at TIMESTAMP,
CONSTRAINT unique_product_warehouse UNIQUE(product_id, warehouse_id),
CONSTRAINT check_reserved_not_exceeds_quantity CHECK (reserved_quantity <= quantity)
);
CREATE INDEX idx_inventory_product_id ON inventory(product_id);
```

Case study

```
CREATE INDEX idx_inventory_warehouse_id ON inventory(warehouse_id);

CREATE INDEX idx_inventory_quantity ON inventory(quantity);

CREATE INDEX idx_inventory_product_warehouse ON inventory(product_id, warehouse_id);

-- Inventory history (audit trail for all changes)

CREATE TABLE inventory_history (

id SERIAL PRIMARY KEY,

product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,

warehouse_id INTEGER NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,

quantity_change INTEGER NOT NULL,

old_quantity INTEGER NOT NULL,

new_quantity INTEGER NOT NULL,

change_type VARCHAR(50) NOT NULL, -- 'purchase', 'sale', 'adjustment', 'transfer', 'initial_stock'

reference_id INTEGER, -- Order ID, Transfer ID, etc.

changed_by INTEGER, -- User ID (would reference users table)

notes TEXT,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);

CREATE INDEX idx_inventory_history_product_id ON inventory_history(product_id);

CREATE INDEX idx_inventory_history_warehouse_id ON inventory_history(warehouse_id);

CREATE INDEX idx_inventory_history_created_at ON inventory_history(created_at);

CREATE INDEX idx_inventory_history_change_type ON inventory_history(change_type);

-- Product bundles (products composed of other products)

CREATE TABLE product_bundles (

id SERIAL PRIMARY KEY,

bundle_product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
```

Case study

```
component_product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE
CASCADE,

quantity_required INTEGER NOT NULL DEFAULT 1 CHECK (quantity_required > 0),

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

CONSTRAINT unique_bundle_component UNIQUE(bundle_product_id, component_product_id),

CONSTRAINT check_no_self_bundle CHECK (bundle_product_id != component_product_id)

);

CREATE INDEX idx_product_bundles_bundle_id ON product_bundles(bundle_product_id);

CREATE INDEX idx_product_bundles_component_id ON product_bundles(component_product_id);

-- Sales/Orders table (for calculating days until stockout)

CREATE TABLE sales_orders (

id SERIAL PRIMARY KEY,

company_id INTEGER NOT NULL REFERENCES companies(id) ON DELETE CASCADE,

warehouse_id INTEGER NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,

order_number VARCHAR(50) NOT NULL,

status VARCHAR(50) DEFAULT 'pending', -- 'pending', 'confirmed', 'shipped', 'delivered', 'cancelled'

order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

shipped_date TIMESTAMP,

total_amount DECIMAL(12, 2),

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

CONSTRAINT unique_order_number_per_company UNIQUE(company_id, order_number)

);

CREATE INDEX idx_sales_orders_company_id ON sales_orders(company_id);

CREATE INDEX idx_sales_orders_warehouse_id ON sales_orders(warehouse_id);

CREATE INDEX idx_sales_orders_order_date ON sales_orders(order_date);

CREATE INDEX idx_sales_orders_status ON sales_orders(status);
```

Case study

```
-- Sales order items (line items)

CREATE TABLE sales_order_items (

id SERIAL PRIMARY KEY,

order_id INTEGER NOT NULL REFERENCES sales_orders(id) ON DELETE CASCADE,

product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE RESTRICT,

quantity INTEGER NOT NULL CHECK (quantity > 0),

unit_price DECIMAL(12, 2) NOT NULL,

total_price DECIMAL(12, 2) NOT NULL,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);

CREATE INDEX idx_sales_order_items_order_id ON sales_order_items(order_id);

CREATE INDEX idx_sales_order_items_product_id ON sales_order_items(product_id);
```

B. Missing Requirements

1. **User Management:**
    a. How do we handle user authentication and authorization?
    b. What roles exist (admin, manager, warehouse staff)?
    c. Can users belong to multiple companies?
2. **Product Bundles:**
    a. How is inventory tracked for bundles? (Virtual vs. Physical)
    b. When a bundle is sold, do we automatically deduct components?
    c. Can bundles contain other bundles (nested)?
    d. What happens if a component is out of stock?
3. **Inventory Transfers:**
    a. Can products be transferred between warehouses?
    b. Do we need approval workflow for transfers?
    c. How do we handle in-transit inventory?
4. **Purchase Orders:**
    a. Do we need to track purchase orders from suppliers?
    b. Should we have automatic reordering when stock is low?
    c. Do we need receiving/goods receipt functionality?
5. **Low Stock Thresholds:**
    a. Can thresholds vary by warehouse for the same product?
    b. Should thresholds be auto-calculated based on sales velocity?
    c. Do different product categories have default thresholds?

Case study

6. **Sales Data:**
    a. Are we tracking actual sales or just inventory changes?
    b. Do we need to differentiate between B2B and B2C sales?
    c. What about returns/refunds?
7. **Pricing:**
    a. Do prices vary by warehouse/location?
    b. Do we need price history tracking?
    c. Currency handling for international companies?
8. **Reserved Inventory:**
    a. How long can inventory be reserved before release?
    b. Can reserved inventory be reallocated?
9. **Warehouse Closure:**
    a. What happens to inventory when a warehouse closes?
    b. Do we need archival vs. deletion?
10. **Performance Requirements:**
    a. Expected number of products per company?
    b. Expected transaction volume?
    c. Real-time vs. batch processing for reports?
11. **Data Retention:**
    a. How long to keep inventory history?
    b. Do deleted products need soft-delete?
12. **Multi-Currency:**
    a. Do we need to support multiple currencies?
    b. Currency conversion handling?

C. Design Decisions & Justifications

1. Multi-Tenant Architecture:

- Every major table includes `company_id` for data isolation
- Ensures Company A cannot access Company B's data
- Composite indexes on `(company_id, other_columns)` for query performance

2. Inventory Table Design:

- Separate `quantity` and `reserved_quantity` fields
- `reserved_quantity` represents orders placed but not yet fulfilled
- Constraint ensures reserved doesn't exceed available
- Unique constraint on `(product_id, warehouse_id)` prevents duplicate entries

3. Inventory History (Audit Trail):

- Immutable log of all inventory changes
- Stores both `old_quantity` and `new_quantity` for easy auditing

Case study

- `change_type` enum for filtering by transaction type
- `reference_id` links to source transaction (order, transfer, etc.)
- Critical for debugging discrepancies and compliance

4. Product-Supplier Many-to-Many:

- Products can have multiple suppliers (backup options)
- `is_preferred` flag indicates primary supplier
- Supplier-specific SKU and pricing tracked
- Enables automatic supplier selection for reorders

5. SKU Uniqueness:

- Global UNIQUE constraint across all companies
- Prevents confusion and simplifies barcode scanning
- Alternative: Could use composite `(company_id, sku)` if companies want same SKUs

6. Decimal for Money:

- `DECIMAL(12, 2)` for all monetary values
- Avoids floating-point precision errors
- 12 digits total, 2 after decimal = up to $9,999,999,999.99

7. Indexes Strategy:

- Foreign keys indexed for JOIN performance
- `company_id` indexed on all multi-tenant tables
- `created_at` indexed for time-based queries
- Composite index on `(product_id, warehouse_id)` for inventory lookups
- `is_active` flags indexed for filtering active records

8. Constraints:

- CHECK constraints prevent negative quantities and prices
- UNIQUE constraints prevent duplicates
- Foreign key CASCADE deletes for child records
- RESTRICT deletes where data must be preserved (order items)

9. Soft Deletes:

- `is_active` flags instead of hard deletes
- Preserves referential integrity
- Allows "undelete" functionality
- Historical reports remain accurate

10. Bundle Design:

- Separate `product_bundles` table for composition
- `is_bundle` flag on products for quick identification
- Self-referencing but prevents circular references
- Question: Does selling a bundle auto-deduct components?

Case study

11. Sales Tracking:

- Full order/order_items structure for accurate sales velocity
- `order_date` indexed for time-series queries
- Status field for order lifecycle
- Necessary for "days until stockout" calculation

12. Timestamp Strategy:

- `created_at` on all tables (immutable)
- `updated_at` on mutable entities
- `last_counted_at` for physical inventory audits
- All timestamps in UTC (application layer converts)

13. Scalability Considerations:

- Partitioning strategy for `inventory_history` by date (future)
- Archive old sales data after 2+ years
- Consider read replicas for reporting queries
- Cache frequently accessed inventory counts

14. Missing: Users/Authentication Table:

- Would need `users` table with company association
- Role-based access control (RBAC)
- Audit fields (`changed_by`) would reference `users.id`

PART 3: API IMPLEMENTATION

A. Assumptions

1. **Low Stock Threshold:** Stored in `products.low_stock_threshold` field (can be overridden per warehouse if needed)
2. **Recent Sales Activity:** Defined as sales within the last 30 days; products with zero sales excluded from alerts
3. **Days Until Stockout Calculation:**
    a. Average daily sales = Total quantity sold in last 30 days / 30
    b. Days until stockout = Current stock / Average daily sales
    c. If average daily sales = 0, return NULL for days_until_stockout

Case study

4. **Multiple Warehouses:** Company can have multiple warehouses; alerts calculated per warehouse independently
5. **Supplier Selection:** Uses preferred supplier (`is_preferred = TRUE`) if available; otherwise, returns first supplier
6. **Alert Criteria:**
    a. Current stock < threshold AND
    b. Product has sales in last 30 days AND
    c. Product is active
7. **Performance:** Assumes database has proper indexes; query optimized for < 1000 products per company
8. **Authentication:** Assumes middleware validates company_id access rights (not implemented here)

B. Implementation

Python

```
from flask import Flask, jsonify, request

from sqlalchemy import func, desc

from datetime import datetime, timedelta

from decimal import Decimal

@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])

def get_low_stock_alerts(company_id):

"""

Returns low stock alerts for all products across company's warehouses.

Filters:

- Current stock < threshold

- Has sales activity in last 30 days

- Product is active

Returns supplier information for easy reordering.

"""
```

Case study

```
try:
# TODO: Verify current user has access to this company
# if not has_permission(current_user, company_id):
# return jsonify({"error": "Unauthorized"}), 403
# Verify company exists
company = Company.query.get(company_id)
if not company:
return jsonify({"error": "Company not found"}), 404
# Date range for "recent sales"
thirty_days_ago = datetime.utcnow() - timedelta(days=30)
# Subquery: Calculate sales velocity (average daily sales per product per warehouse)
# This gets total quantity sold in last 30 days
sales_velocity_subquery = db.session.query(
sales_order_items.c.product_id,
sales_orders.c.warehouse_id,
func.sum(sales_order_items.c.quantity).label('total_sold')
).join(
sales_orders,
sales_order_items.c.order_id == sales_orders.c.id
).filter(
sales_orders.c.company_id == company_id,
sales_orders.c.order_date >= thirty_days_ago,
sales_orders.c.status.in_(['confirmed', 'shipped', 'delivered']) # Exclude cancelled
).group_by(
sales_order_items.c.product_id,
sales_orders.c.warehouse_id
```

Case study

).subquery('sales_velocity')

Main query: Get inventory below threshold with sales activity

alerts_query = db.session.query(

Product.id.label('product_id'),

Product.name.label('product_name'),

Product.sku,

Product.low_stock_threshold.label('threshold'),

Inventory.warehouse_id,

Warehouse.name.label('warehouse_name'),

Inventory.quantity.label('current_stock'),

sales_velocity_subquery.c.total_sold,

Supplier.id.label('supplier_id'),

Supplier.name.label('supplier_name'),

Supplier.contact_email.label('supplier_email'),

ProductSupplier.lead_time_days.label('supplier_lead_time')

).join(

Inventory,

Product.id == Inventory.product_id

).join(

Warehouse,

Inventory.warehouse_id == Warehouse.id

).outerjoin(

sales_velocity_subquery,

db.and_(

Product.id == sales_velocity_subquery.c.product_id,

Inventory.warehouse_id == sales_velocity_subquery.c.warehouse_id

Case study

```
)
).outerjoin(
ProductSupplier,
db.and_(
Product.id == ProductSupplier.product_id,
ProductSupplier.is_preferred == True # Get preferred supplier
)
).outerjoin(
Supplier,
ProductSupplier.supplier_id == Supplier.id
).filter(
Product.company_id == company_id,
Product.is_active == True,
Warehouse.is_active == True,
Inventory.quantity < Product.low_stock_threshold, # Below threshold
sales_velocity_subquery.c.total_sold > 0 # Has recent sales
).order_by(
# Prioritize by urgency: lowest stock percentage first
(Inventory.quantity.cast(db.Float) / Product.low_stock_threshold.cast(db.Float))
)
# Execute query
results = alerts_query.all()
# Format response
alerts = []
for row in results:
# Calculate days until stockout
```

Case study

```
# Average daily sales = total sold / 30 days

avg_daily_sales = row.total_sold / 30.0 if row.total_sold else 0

# Days until stockout = current stock / average daily sales

if avg_daily_sales > 0:

days_until_stockout = int(row.current_stock / avg_daily_sales)

else:

days_until_stockout = None # No recent sales, can't calculate

# Build alert object

alert = {

"product_id": row.product_id,

"product_name": row.product_name,

"sku": row.sku,

"warehouse_id": row.warehouse_id,

"warehouse_name": row.warehouse_name,

"current_stock": row.current_stock,

"threshold": row.threshold,

"days_until_stockout": days_until_stockout,

"supplier": None

}

# Add supplier info if available

if row.supplier_id:

alert["supplier"] = {

"id": row.supplier_id,

"name": row.supplier_name,

"contact_email": row.supplier_email,

"lead_time_days": row.supplier_lead_time
```

Case study

```
}

else:

# No preferred supplier, get any supplier for this product

fallback_supplier = db.session.query(

Supplier.id,

Supplier.name,

Supplier.contact_email,

ProductSupplier.lead_time_days

).join(

ProductSupplier,

Supplier.id == ProductSupplier.supplier_id

).filter(

ProductSupplier.product_id == row.product_id,

Supplier.is_active == True

).first()

if fallback_supplier:

alert["supplier"] = {

"id": fallback_supplier.id,

"name": fallback_supplier.name,

"contact_email": fallback_supplier.contact_email,

"lead_time_days": fallback_supplier.lead_time_days

}

alerts.append(alert)

# Build response

response = {

"alerts": alerts,
```

Case study

```
"total_alerts": len(alerts),
"company_id": company_id,
"generated_at": datetime.utcnow().isoformat() + "Z"
}
return jsonify(response), 200
except Exception as e:
# Log error
app.logger.error(f"Error generating low stock alerts for company {company_id}: {str(e)}")
return jsonify({
"error": "Failed to generate alerts",
"message": "An internal error occurred. Please try again later."
}), 500
# Optional: Add query parameter filtering
@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def get_low_stock_alerts_enhanced(company_id):
"""
Enhanced version with optional query parameters:
- warehouse_id: Filter by specific warehouse
- category: Filter by product category
- min_urgency: Only show alerts with days_until_stockout <= min_urgency
"""
# Get query parameters
warehouse_id = request.args.get('warehouse_id', type=int)
category = request.args.get('category', type=str)
min_urgency_days = request.args.get('min_urgency', type=int)
# [Same logic as above, but add filters based on query params]
```

Case study

This allows users to narrow down alerts as needed

pass # Implementation similar to above with additional filters

C. Edge Cases Handled

1. Company Doesn't Exist:

Query returns 404 with clear error message

Prevents unnecessary database operations

2. No Warehouses for Company:

Query returns empty alerts array with total_alerts: 0

Valid response, not an error condition

3. Products with No Sales History:

Filtered out by sales_velocity_subquery.c.total_sold > 0 condition

Rationale: Can't predict stockout date without sales data

Alternative: Could show these separately as "new products"

4. Products with No Supplier:

OUTER JOIN allows products without suppliers

Response shows `"supplier": null